Preprint ANL/MCS-P1573-0109, Math and Comp. Sci. Division, Argonne Natl. Lab.

CANONICAL NUMBERING SYSTEMS FOR FINITE-ELEMENT CODES

Timothy J. Tautges

Argonne National Laboratory, Argonne, IL tautges@mcs.anl.gov

ABSTRACT

Canonical numbering systems are used to relate finite-element vertices and elements to edges and faces in those elements. A numbering system is proposed that treats all the major topological element types used in practice. Also described is a set of functions that provide common evaluations of the canonical numbering data. Examples from various parts of the finite-element analysis process are used to show the usefulness of these functions. The differences between various numbering systems used in commercial and research codes and our numbering system are described. The implementation of these functions is available as open-source software and can be called directly from the C, C++, and Fortran languages.

1. INTRODUCTION

Scientific computing has been profoundly changed by software components, most clearly in areas such as numerical linear algebra solvers [BAL04] and mesh partitioning [MET08], but elsewhere as well. The shift to component-based computing has enabled the rapid incorporation of the latest technology into applications (i.e., “vertical” integration) and has simplified exploration of available components for a given technology to allow selection of the one most suitable for the application (i.e., “horizontal” integration). Following this trend, components are now being developed for mesh generation and other mesh-based enabling technologies [ITA08,TAU04]. We expect similar gains in functionality and sophistication of applications based on these new mesh-based components.

Applications and mesh-based components using mesh data must have a common understanding of the mesh in order to work correctly. For example, the ordering of vertices in an element is important in uniquely describing that element. Figure 1 shows two possible numberings of vertices in a tetrahedron.

If the element volume is defined in an application as the vector triple product,

$$V = \left(\overrightarrow{01} x \overrightarrow{02}\right) \cdot \overrightarrow{03}$$

then one of the numberings will generate a positive volume for this element, while the other will result in a negative volume. Applications must either use a common numbering system when evaluating the mesh or at least know how to translate between various numbering systems if several are used. Although there is wide agreement on the basic element types in the finite element “zoo” (e.g., nodes, triangles, hexahedra), differences in the numbering definitions present barriers to interoperability between components using these data. Enumerating the numbering differences between common numbering systems, as well as facilitating the translation between the various systems, would improve interoperability and contribute to the deployment of mesh-based components.

This paper defines a canonical numbering system, referred to as MBCN (MoaB Canonical Numbering), that is comprehensive, covering most known element types used in practice, while also being extensible to new element types. Convenience functions are described that depend on, but are not strictly part of, a minimal numbering definition. Such functions often are implemented in mesh-based applications, but in a form that usually does not fully support a complete set of element types. A functional interface for accessing the numbering definition is described, along with an implementation that can be accessed from C, C++, and Fortran applications. Mappings between the numbering system proposed here and several other popular numbering systems, including those used in PATRAN [PAT08], ExodusII [SCH92], and Ansys [ANS98], are also described. These functions have been released under the open source LGPL license and are available for download from the web [MOA08].

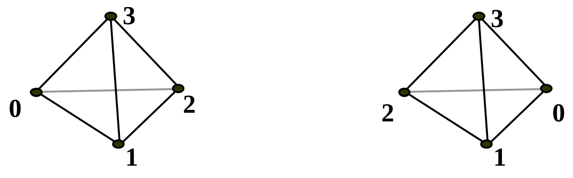

Figure 1: Two different numberings for a tetrahedron.

The Interoperable Tools for Advanced Petascale Simulations (ITAPS) project develops interfaces to geometry, mesh, and field data and services based on these interfaces [ITA08]. The ITAPS mesh interface, referred to as iMesh, uses a set of entity types similar, but not identical, to those defined in MBCN. The Mesh-Oriented datABase library (MOAB) implements the iMesh interface specification and is tuned for minimal memory use first and execution time second [TAU04]. The entity types treated by both MOAB and MBCN are identical. ITAPS references the work described in this paper for its canonical numbering, with MBCN’s set of entity types being a superset of those defined by iMesh. Because MBCN defines entities not treated by iMesh, and because of other small differences (e.g., MBCN’s defining an “entity set” as an entity type and its handling of higher-order nodes in quadratic elements), MBCN is reported separately from iMesh.

In Section 2 we describe the canonical numbering system used in MBCN. In Section 3, we describe some higher-level functions for evaluating canonical numbering and show how such functions are often used implicitly in codes that evaluate the mesh. The implementation of MBCN minimizes the actual data used to describe the numbering system, thereby minimizing the chances of consistency errors and simplifying the inclusion of different element types; this implementation is described in Section 4. Also described are MBCN functions for translation between various systems using permutation vectors. A functional interface specification for MBCN is given in Appendix A. The MBCN numbering system is related to other popular numbering systems, including those used in Ansys and Sandia's ExodusII format, in Appendix B.

2. MBCN NUMBERING SYSTEM DEFINITION

We divide the definition of our canonical numbering system into several logical parts. First, the types of elements supported by the system are enumerated. Next, the numbering of vertices, edges, and faces in each element is defined. Third, the process for numbering so-called higher-order vertices in quadratic elements is defined.

2.1. MBCN ELEMENT ZOO

The finite-element zoo treated by MBCN is shown in Figure 2. This set includes all of the commonly used elements in finite-element analysis, including triangles, quadrilaterals, tetrahedra, and hexahedra. The set also includes some of the less common elements, including wedges (also referred to as triangular prisms) and pyramids. Because these elements are less common, the numbering systems used to describe them often vary. By including them in our system, we hope to popularize a numbering convention for these elements. Some special-purpose elements also are included in our system (e.g., the "knife" element [MOA08]), for the same reasons. Additionally, our enumeration of element types includes some types not traditionally found in the finite element zoo, including polygon, polyhedron, and entity set. We have included these types in the enumeration for convenience, noting that iteration is often done over all element types, not only those that have a defined numbering (for more information, see [MOA08]).

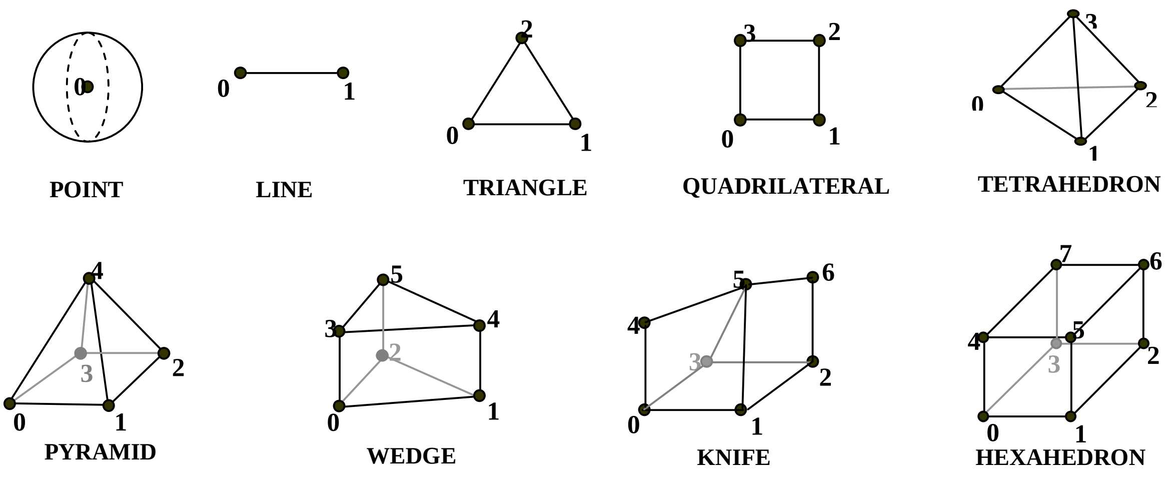

Figure 2: Vertex numbering for finite-element zoo treated in MBCN.

Besides containing an enumeration of the types defined there, MBCN has functions that return a string name for a given type and the type given a string name. Full specifications for these functions are given in Appendix A.

2.2. VERTEX, EDGE, AND FACE NUMBERING

A complete vertex numbering definition for these elements is shown in Figure 2. Edge and face numberings are shown in Figure 3 and Figure 4, respectively. One point to note is that all vertex, edge, and face numbers within each element are contiguous and begin with zero.

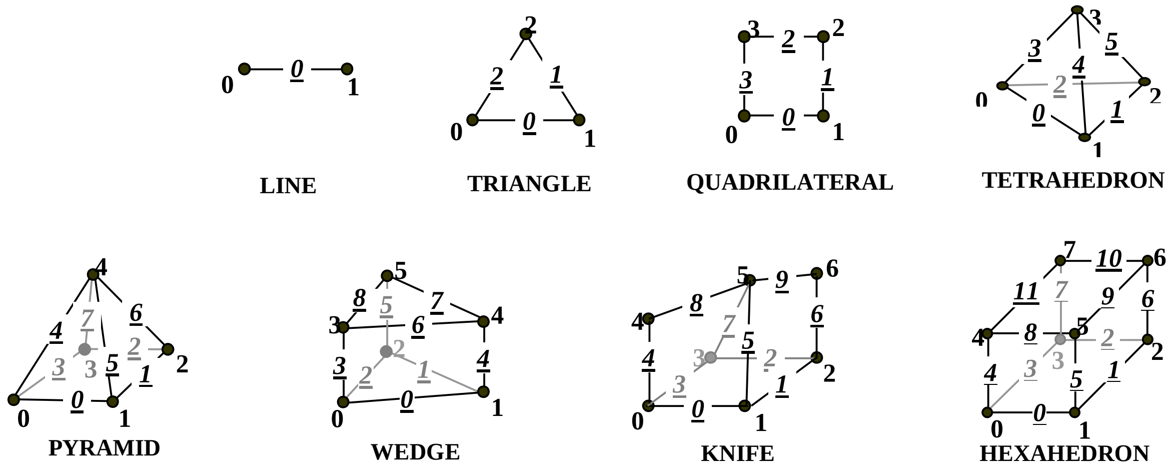

Figure 3: Edge numbering for finite-element zoo treated in MBCN.

Faces and edges bounding elements of higher topological dimension are referred to as "facets" of those elements; vertices can also be considered facets, although they are not strictly included in that definition in the topology literature. MBCN has functions that return the number of facets of each dimension d < D, for elements of dimension D. MBCN also has functionality to convert the base of the numbering system from zero to one. See Appendix A for details.

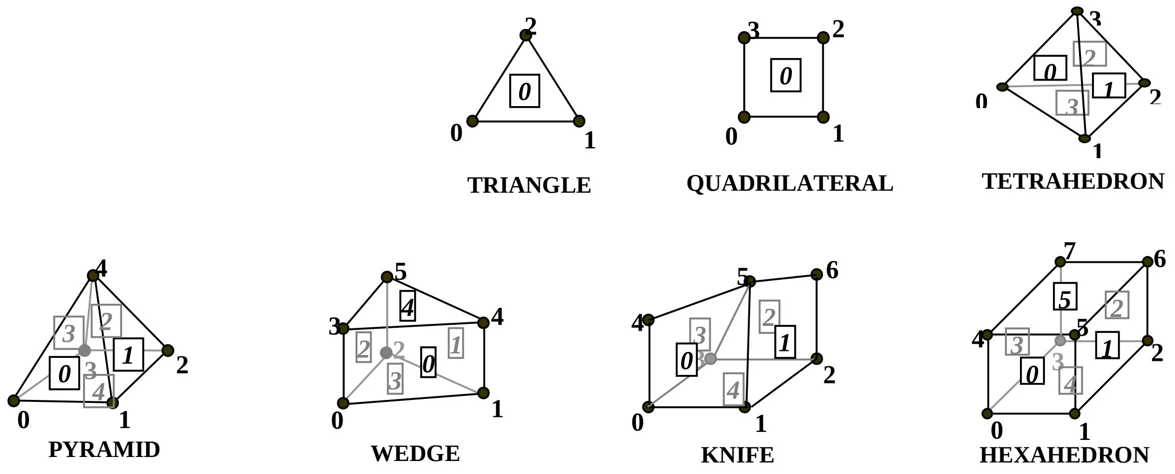

Figure 4: Face numbering for finite-element zoo treated in MBCN.

2.3. HIGHER-ORDER ELEMENTS

In its most general form, the finite element method can use a variety of shape function types of arbitrary order. In practice, however, the majority of finite-element analysis codes, especially in the solid mechanics community, use linear or quadratic shape functions. From the viewpoint of the mesh, quadratic shape functions are supported by adding degrees of freedom to some or all of the lower-dimensional facets and to the element itself, one for every order beyond linear. The two most common methods for representing these degrees of freedom are as attributes assigned directly to those facets or as “higher-order” vertices associated indirectly with the facets they resolve. The former representation is more general, allowing each facet to have an arbitrary number of degrees of freedom. It is also more costly, however, in terms of memory, since lower-dimensional facets must be explicitly represented [BEA97]. In cases where all facets of a given topological dimension have the same number of degrees of freedom, the second method is easily represented by concatenating higher-order vertices to the list of corner vertices. This method is used by the majority of numbering systems reviewed in this paper (see Appendix B).

In many cases, some but not all lower dimensions are resolved by extra degrees of freedom. For example, the widely used 10-vertex tetrahedron has mid-edge as well as corner vertices, but not mid-face vertices. If a flag h_d is used to specify whether facets of dimension d are resolved with higher-order vertices, and n_d is the number of facets of dimension d, then the total number of vertices in an element of dimension D is given by

$$N = \sum_{d=0}^{D} h_d n_d$$

(we assume n_0 is the number of corner vertices; h_0 is always unity). For example, for a 10-vertex tetrahedron with corner and mid-edge vertices, h is [1, 1, 0, 0], n is [4, 6, 4, 1], and N = 4 + 6 =

[4]By definition, the MBEntityType enumeration begins with the value of zero for MBVERTEX and ends with the value MBMAXTYPE value.

10. Similarly, a hexahedron with higher-order vertices resolving the edges and the element itself would have h = [1, 1, 0, 1], n = [8, 12, 6, 1] and N = 21. One subtle aspect of this numbering approach is that the index of a higher-order vertex may have different values depending on which facet dimensions of an element are resolved with higher-order vertices. For example, the index of the vertex resolving the interior of the hexahedron would change from 20 to 26 if h were changed to [1, 1, 1, 1] (assuming a zero-based numbering system). Thus, a "canonical" ordering for higher-order vertices depends on which facet dimensions are resolved in a higher-order element.

All possible combinations of h_d for a given dimension, along with the topology of a given element type, can be used to enumerate the possible total number of vertices for that element. If these numbers are unique for a given element, then h_d can be inferred given the element type and the total number of (corner and higher-order) vertices N. Going the other way, h_d and N can be used to find the index and dimension of facet resolved by vertex I. Both operations are common in meshing applications.

Functions performing these types of evaluations are described further in Appendix A.

3. FUNCTION INTERFACE TO CANONICAL NUMBERING DATA

MBCN is implemented as a C++ class and does not depend on the rest of MOAB; therefore, it can be incorporated as-is into applications if desired. Building MBCN requires files generated from the Automake-based MOAB build process, although this would be easy to adapt to an application's build system. Most functions in MBCN are implemented as inline functions, with basic numbering data stored in fixed-size static arrays. Thus, calls to MBCN are efficient in run-time.

MBCN also provides wrapper functions that can be called from C and Fortran; these functions use a slightly different syntax from that of the MBCN class functions. Wrapper functions have "MBCN_" prepended to the function name. These functions are all declared with void type; those corresponding to MBCN class functions that return non-void values have an extra argument of that type in their C/Fortran counterpart. Default arguments always appear in the argument list of wrapper functions. Moreover, any C++ data structures not found in C/Fortran (e.g., Standard Template Library vectors std::vector< >) are passed as an array and a size in the wrapper functions. Other details of the C-Fortran interface can be found in the MBCN header file.

There are basic functions needed to access the data shown in Figures 2– 4; these functions are described in the following subsection. Convenience functions can also be defined, which depend on the basic functions but provide higher-level functionality (e.g., adjacency evaluation) in terms of canonical numbering data. These functions are described following the basic functions. In both cases, only a qualitative description is given here, with more complete descriptions provided in Appendix A.

3.1. BASIC CANONICAL NUMBERING FUNCTIONS

The following functions give information about the entity types defined by MBCN.

- **MBEntityType:** Denotes entity types. Arguments of this type are input to most of the other MBCN functions. The types used by MBCN are those shown in Figure 2, along with MBENTITYSET and MBMAXTYPE.
- **EntityTypeName(t):** Returns a constant character string representing the name of type **t**.
- **Dimension(t):** Returns the topological dimension of this entity type.

The following functions return information about the intrinsic numbering data for entities, that is, the numbering data that cannot be derived from other numbering information for an entity:

- **VerticesPerEntity(t):** The number of corner vertices for entities of type **t**.
- **NumSubEntities(t, d):** The number of bounding facets of dimension **d** for entities of type **t**.
- **SubEntityType(t, d):** The type of bounding facet of dimension **d** for entities of type **t**; for d != 2, the returned type will always be the same (MBEDGE for d = 1, MBVERTEX for d = 0, and t for d = Dimension(t)); for d = 2, some entity types (e.g., MBPRISM) have facets of more than one type.
- **SubEntityConn(t, d, n, ...):** The connectivity of the **n**th facet of dimension **d** bounding an entity of type **t**. Connectivity is returned in terms of the corner vertex numbers of the entity of type **t**.

These functions are sufficient for describing the entities shown in Figures 2-4. Existing applications usually have some version of this information already implemented, at least for the types of elements they currently use. New applications could use these functions instead of implementing them themselves.

3.2. HIGHER-LEVEL CONVENIENCE FUNCTIONS

The fundamental data in a canonical numbering definition is accessed by using the functions described above. In addition to those data and functions, however, certain higher-level functionality that depends on canonical numbering is also commonly implemented in mesh-based applications. MBCN provides some of these functions, which apply to all treated element types, in contrast to the more limited implementations commonly found in applications. Examples of such functionality are discussed in this section.

Facet Adjacencies

Many codes represent elements by using only an array of the vertices defining the element, but need information about facets of these elements. For example, a code might need to find the vertices common to two faces in the element. The AdjacentSides function evaluates adjacencies based on canonical numbering data, returning the results in terms of those same data.

```
int AdjacentSides(const MBEntityType t, const int *i, const int num_i,

                  const int source_dim, const int target_dim,

                  std::vector<int> &index_list,
                                    const int operation_type = MBCN::INTERSECT)
```

Given an entity type *t*, a vector of facet indices (*i*) of specified dimension (source_dim), and the target dimension, this function returns the indices of facets of that target dimension shared by the source facets. If operation_type is specified as UNION instead of INTERSECT, any target facets adjacent to the source facets are included in the output list. Note that this function can traverse upward or downward in dimension; for example, this function can be used to find the faces sharing a given edge or the vertices common to two faces.

The following is the equivalent function in the C/Fortran wrapper.

```
void MBCN_AdjacentSides(int t, int *i, int num_i,
                  int source_dim, int target_dim,
                  int *index_list, int index_list_size,
                  int operation_type, int *num_sides_returned)
```

Example usages of the AdjacentSides function, used in various places in MOAB, include the following.

- Find the vertex indices defining a specified facet of an element (some mesh formats, e.g., Ansys, can specify element facets in terms of the vertices defining those facets).

  ```
  AdjacentSides(t, &side_no, 1, side_dim, 0, side_vertices)
  ```

- Find all edges shared by the faces in face_list.

  ```
  AdjacentSides(t, face_list, num_faces, 2, 1, common_edges)
  ```

- Find all faces sharing the apex node of a pyramid.

  ```
  AdjacentSides(MBPYRAMID, &int(4), 1, 0, 2, sharing_faces, MBCN::UNION)
  ```

Translation of Facet Connectivity to Facet Index

Many mesh generation codes represent boundary conditions by using the geometric model entities, while the mesh storage format calls for boundary condition data in terms of "sides", or local facet indices in an element. For example, the Exodus [SCH92] format stores Neumann-type boundary conditions as "sidesets," which are a collection of pairs of elements and facet numbers. In these cases the code must find the facet number given that facet and the element it bounds. For this purpose, the following function is provided in MBCN:

```
int SideNumber(const X *parent_conn, const MBEntityType parent_type,
               const X *facet_conn, const int facet_num_verts,
               const int facet_dim, &side_number, int &sense, int &offset)
```

This function takes connectivity in generic form; in the actual implementation, several versions of this function are overloaded with the same name and vary according to the data type of the arguments. In addition to returning the facet number, the sense of the facet (i.e., the normal indicated by the order of vertices in facet_conn compared to that of the facet in the canonical ordering in parent_conn) and the offset of the start of facet_conn compared to that of the canonical ordering data are also returned. Returning these items incurs no cost inside the implementation because the items are computed as part of finding the facet number.

For example, the facet number of a quadrilateral with vertices in face_connect in a hexahedron whose vertices are in hex_connect can be found by calling the following.

```
SideNumber(hex_connect, MBHEX, face_connect, 4, 2, side, sense, offset)
```

Higher-Order Vertex Indexing

As discussed in Section 2.3, a common operation is to find the facet resolving a vertex, given the element type and the total number of vertices; this operation is performed by the function

```
void HONodeParent(const MBEntityType elem_type,

                  const int num_verts, const int ho_vert_index,

                  int &parent_dim, int &parent_index)
```

where the dimension and index of the resolved facet are returned in the parent_dim and parent_index arguments, respectively.

Permutations and Relation to Other Numbering Systems

In order to maximize compatibility with other codes, the canonical numbering system defined by MBCN corresponds to the PATRAN numbering system [PAT08] in most cases. However, MBCN supports several element types not supported by PATRAN; in those cases, if a corresponding element type is available in ExodusII [SCH92], that definition is used. Otherwise, MBCN defines its own numbering.

Besides being useful for implementation of various functionality within a mesh application, canonical numbering information is useful for translating mesh between various mesh storage formats. Using the correct ordering for the originating and destination formats is critical for obtaining valid mesh definitions on each side of the translation. Canonical numbering can be translated by using a vector to permute the data from MBCN's numbering system to the target system, or vice versa. MBCN allows applications to define permutation vectors based on entity type, facet dimension, and number of entities of that dimension and to permute index lists based on those vectors. The following functions are provided for this purpose.

- **setPermutation(t, d, ilist, n, f), resetPermutation(t, d):** sets and resets the permutation vector for entities of type **t**, facet dimension **d**, and number of sub facets **n**, to the vector in **ilist**. If f is non zero, **ilist** contains the reverse-permutation vector**. n** is provided to allow setting vertex numbering for higher-order elements, where the number of vertices depends on the particular kind of element (see Section 2.3).

- **permuteThis(t, d, conn, ni, ne), revPermuteThis(t, d, ilist, ni, ne):** for entities of type **t**, dimension **d**, number of indices per facet **ni**, and number of entities **ne**, permutes (converts from MOAB to application-specified order) or reverse-permutes (converts to MOAB order from application-specified order) in-place the indices in **ilist**.

The data in ilist can be either canonical number index lists or application-dependent entity handles. For the latter case, overloaded versions of the permuteThis and revPermuteThis functions are available for integer, unsigned integer, long integer, and void* data.

It can be challenging to track down the numbering systems used in various finite-element codes, commercial or otherwise, and even more challenging to find differences between these systems. For reference, we describe the numbering systems used in several well-known codes, including PATRAN, in Appendix B of this paper.

4. COMPLEXITY AND EFFICIENCY

For practical use, translation of numbering must be efficient in terms of both memory and run-time. The two main data arrays, for downward and upward connectivity numbering, require about 10 kB and 60 kB, respectively. This amount of memory is almost negligible in comparison to overall system memory today.

Applications will be most sensitive to run-time in cases where the entire fine-grained data set is being processed. This is most likely to be the case when element numbering is permuted from one system to another. For this reason, the permutation functionality discussed in Section 3.2 uses index arrays for both reverse and forward permutations. The run-time complexity of permuting based on these arrays is therefore linear in the size of the connectivity array being permuted. Permutation requires almost no extra memory, since connectivity is permuted one element at a time, then copied back to the original connectivity array passed in by the application.

The next most likely place where run-time might be an issue is in the translations between facet connectivity and index, discussed in Section 3.2. In this case, for each entity/facet pair, MBCN find the index of each facet vertex in the entity's connectivity array, then compares that facet index array against the connectivity arrays of sub facets of the facet dimension. Although involving traversal of several arrays in this process, these arrays are all relatively short and are bounded by either the maximum number of vertices in an entity (currently 8 in MBCN) or sub facets bounding an entity (12). Furthermore, this type of operation usually occurs as part of the input/output process and therefore is amortized over the run-time for the whole application. For example, a one-million element hexahedral mesh was generated for a standard brick shape, with six "sideset" boundary condition sets. This mesh was written from MOAB's representation, based on elements and facets, to an Exodus file, based on element and facet numbers. Finding the facet indices required only 1.6% of the overall time to write this file. We conclude that run-times of this sort will not be significant for most applications.

5. SUMMARY

This paper describes a system for numbering vertices and facets of elements in a finite-element mesh. The system corresponds closely to the PATRAN and EXODUSII numbering systems where possible, but it also includes elements not supported in those systems. Numbering for both linear and quadratic elements is described, including cases where only some lower dimensions are resolved with nodes (e.g., a 10-node tetrahedron containing refined edges but not faces). Functions are also described for evaluating canonical numbering data. These functions are used to accomplish tasks often found in mesh applications. MBCN provides these functions for all supported element types, a feature that can simplify enhancements to applications to support new element types.

The numbering system and evaluation functions described in this paper are implemented in the MBCN class, which has been released as open source software. Functions are also provided that can be called directly from the Fortran and C languages. MBCN uses static arrays to store numbering information and adds less than 100 kB to an application. In most cases, run-time complexity is linear or less in the number of vertices in a connectivity array.

This paper also relates our numbering system with that of other common systems, including PATRAN, ExodusII, and Ansys. Functions provided by MBCN facilitate translation between the various numbering systems. The run-time complexity of permutations is minimized by permuting in-place for single elements using small static arrays. By providing both code and translations between various numbering systems, MBCN facilitates interoperability between various mesh-related applications. Interoperability has been shown to increase the pace of development of scientific computing applications.

ACKNOWLEDGMENTS

This work was supported by the US Department of Energy's Scientific Discovery through Advanced Computing program under Contract DE-AC02-06CH11357. Argonne National Laboratory is managed by UChicago, Argonne LLC under Contract DE-AC02-06CH11357.

APPENDIX A. API FOR QUERYING CANONICAL ORDERING

Given a canonical ordering specification like the one in this paper, an Application Programming Interface (API) is also useful for defining functional interfaces used to access numbering information. Functions and enumerated types for returning various numbering information are described briefly in this appendix; full function definitions and reference information can be found in the MBCN header file. For convenience, the API description is grouped into several sections:

- Enumerated types and miscellaneous functions
- Basic data functions
- Evaluation functions
- Functions operating on application data
- Higher-order node functions

A1. ENUMERATED TYPES AND MISCELLANEOUS FUNCTIONS

MBCN references entity types using values of an enumerated type. A few functions also get or set variables not specific to any entity type. These types and functions are summarized in .

A2. BASIC DATA FUNCTIONS

Basic data functions return numbering information that depends only on the entity type and any sub-entity type or index for which numbering information is being requested. This data can be considered statically defined and is not likely to change in subsequent versions. The basic data functions are summarized in .

A3. DATA EVALUATION FUNCTIONS

Data evaluation functions provide information derived from the static canonical numbering data, but not necessarily stored in static tables. These functions may or may not be as efficient as those in the previous section.

A4. FUNCTIONS OPERATING ON APPLICATION DATA

Canonical numbering functions are often used to find numbering information on specific application data. For example, given the connectivity array for an element and a vertex number, find the index of that vertex in the element. The functions in this section are based on two important assumptions:

- The data in the connectivity array are sizeof(void*) on the computer being used.
- A compare operator is defined that determines whether two index values represent the same index.

If these assumptions are correct, applications can call these functions, passing in connectivity arrays as-is. Casting is used inside the implementation to avoid dereferencing void pointers. summarizes the MBCN functions that operate on application data.

A5. FUNCTIONS INVOLVING HIGHER-ORDER NODES

The indices of higher-order nodes in an element depend dynamically on the other higher-order nodes defined for the element. If one stipulates that if any subentities of a given dimension are resolved with high-order nodes, then all must be, indices for higher-order nodes can be computed with little additional information. The functions summarized in provide indexing information for entities with higher-order nodes.

APPENDIX B. MAPPINGS FROM OTHER SYSTEMS

Various well-known canonical numbering systems are used in the community, for example, PATRAN, ExodusII, and STEP. To our knowledge this is the first attempt to publicly document and compare these numbering systems. In general, each numbering system is the result of having to support legacy applications as well as achieving some sort of consistency; often these goals are competing. In most cases there is no clear “best” way to construct this numbering. In such situations, it is most useful to simply understand how each numbering system differs from the others.

The point of comparison used in this section is the numbering system defined in Section 2.

B1. EXODUSII [SCH92]:

TET:

- Face-resolving higher order nodes appear in the connectivity list in the order f1, f2, f4, f3, after any edge-resolving nodes and before a tet-resolving node.

HEX:

- In the 27-node hex (i.e. a hex with all dimensions resolved with higher-order nodes), the hex-resolving node appears before the face-resolving nodes. That is, nodes appear in the order (v1-v8, e1-e12, h1, f1-f6).

B2. PATRAN [PAT08]:

TET:

- Edge numbering begins with base edges in order consistent with vertices, then proceeds to side edges in order consistent with edge ordering around base face.
- Face numbering begins with base face, then proceeds to side faces in order consistent with base edges.
- Base face has inward normal; others have outward normal.
- Higher-order node numbering is consistent with edge numbering, but higher-order node indices for face-resolving nodes are ordered f4, f2, f3, f1.

PRISM:

- In PATRAN these are named “WEDGE.”
- Edge numbering begins with base edges, proceeding to top and then side edges.
- Face ordering is consistent with edge ordering.
- Base face has inward normal; others have outward normal.
- Higher-order node numbering is not consistent with edge ordering.

PYRAMID

- PATRAN has no pyramid element.

HEX:

- Face ordering begins with base and top faces, then lists side faces in consecutive order around the first face.
- Edge ordering is consistent with face ordering.
- Base face has inward normal; others have outward normal.
- Higher-order node ordering is not consistent with edge numbering.

B3. STEP 10303-104

TET

- Face numbering begins with the base face, proceeding to side faces in order consistent with edge ordering around the base face.
- Higher-order node numbering is inconsistent with edge ordering, beginning with side edges and proceeding to the base edges.

PRISM

- STEP 10303-104 refers to this element as a “wedge.”
- Edge numbering begins with base edges, followed by the top edges, followed by side edges; base, top, and side edge ordering is consistent with vertex numbering on base face.
- Face ordering is consistent with edge ordering.
- Higher-order node numbering begins with side edge mid-nodes, then base and top edge mid-nodes; it is inconsistent with edge and face ordering.

PYRAMID

- Face ordering begins with the base face with outward normal, then proceeds to side faces in order consistent with edge ordering on the base face.
- Higher-order node numbering begins with side edges, then proceeds to the base edges; it is inconsistent with edge and face numbering.

HEX

- Edge ordering begins with the base edges, then proceeds to top and then side edges.
- Face ordering begins with base and top faces, then proceeds to side faces, in order to be consistent with edge ordering around the base face.
- Higher-order node numbering begins with side edges, then proceeds to base and top edges, then to faces in similar order; it is inconsistent with edge and face numbering.

B4. OTHER FORMATS

This section contains general comments about the compatibilities or incompatibilities of other numbering systems used in the finite-element community.

Fluent/Gambit

- Vertex and face numberings on two-dimensional elements follow a left-hand rule.
- Vertex and edge numbering is consistent with PATRAN for some 3D elements, not for others (generally, hex elements have unique numbering)
- Higher-order nodes are interleaved with corner nodes for higher-order elements, rather than appearing after all corner nodes.

Ansys

- Vertex and apparent edge numberings are similar to those in PATRAN.
- Face numbering generally goes bottom-side-top, which is different from most others.
- Higher-order node numbering follows edge and face numbering.
- PRISM and PYRAMID elements seem to be supported only using degenerate hex elements.

Abaqus

- Vertex, apparent edge and face numberings similar to those in PATRAN.
- Higher-order node numbering follows edge/face numbering.

APPENDIX C. MBCN IMPLEMENTATION

MBCN can be divided into two parts: fundamental canonical numbering data for the supported element types, and functions for evaluating those data to provide higher-level capability (like that described in the previous section). The challenge in implementing a canonical ordering system is not in the basic implementation; rather, it is in doing the implementation in a way that reduces the possibility of consistency errors and simplifies adding new element types. The MBCN implementation meets both these goals, possibly at the expense of code clarity. Therefore, we describe the implementation of the fundamental canonical ordering data here (implementation of the higher-level functions is relatively straightforward and is not described here).

C1. FUNDAMENTAL DATA

Canonical numbering can be thought of as the relation between the vertices describing a *D*-dimensional element and those forming its *d*-dimensional bounding facets, $0 < d < D$. Working in that paradigm, we store canonical numbering data in an array `mConnectivityMap[t][j]` of `ConnMap` structures, where[4] t = `MBVERTEX..MBMAXTYPE-1`, and j = 0..2 (j=0 for edge facets, j=1 for face facets, and j=2 for the element itself). The following data are stored in the `mConnectivityMap[t][j]` corresponding to element type t and facet dimension $j+1$:

- `int topo_dimension`: d, the topological dimension of element type t

- `int num_sub_elements`: number of facets of dimension *j+1* (j=0..*d*-2) or unity (*j=d-1*)
- `int num_nodes_per_sub_element[i]`: number of vertices contained in facet *i* of dimension *j+1* (*j=0..d-2*) or number of corner vertices in elements of type *t* (*j=d-1, i=0*)
- `MBEntityType target_type[i]`: element type of facet *i* (*j=0..d-2*) or element type *t* (*j=d-1, i=0*)
- `int conn[i][]`: connectivity of facet *i*, in terms of vertex indices in element type *t* (*j=0..d-2*), or indices 0..`num_nodes_per_sub_element[0]` (*j=d-1*)

Note that facet information is not needed for facets of dimension zero (i.e., vertices) and that the `ConnMap` structures for a given type *t*, `mConnectivityMap[t][]`, have a small amount of redundant data, for convenience.

Using the data stored in `mConnectivityMap` as described above, one can read many of the basic canonical numbering data right from the table, for example, the following.

- Topological dimension of elements of type *t* = `mConnectivityMap[t][0].topo_dimension`
- Number of (corner) vertices for element type *t* = `mConnectivityMap[t][mConnectivityMap[t][0]].. num_sub_elements`
- Number of facets of dimension *d* for element type *t* = `mConnectivityMap[t][d-1].num_sub_elements`

These functions and others are implemented as inline functions that simply reference the appropriate items in `mConnectivityMap`.

C2. UPWARD ADJACENCY SUPPORT

As described in the preceding section, the AdjacentSides function can be used to find facet indices adjacent to a given set of facets. This function can also perform both intersections and unions of facet indices for a given set of source facets. To support this function, MBCN also defines a parallel table of adjacency information in the variable `mUpConnMap`. Whereas `mConnectivityMap[`*t*`][`*j*`]` stores data pertaining to facets of *lower* dimension *j*, `mUpConnMap[`*t*`][`*j*`][`*k*`]` stores adjacency data pertaining to source and target dimensions *j* and *k*, respectively, *j* < *k*. The facet connectivity data stored in `mConnectivityMap[`*t*`][`*j*`].conn[`*j*`][]` is stored in canonical numbering order of the facet, while the adjacent facet indices stored in `mUpConnMap` are sorted by index value; this approach facilitates intersection and union operations supported by the AdjacentSides function.

For a more detailed discussion of this issue, see [TAU04].

C3. EXTENDING MBCN

Extending MBCN would be accomplished by adding data for a new element type to the `mConnectivityMap` and `mUpConnMap` static arrays (along with the new element type itself). Since the other functions in MBCN are implemented based on those arrays, they would account for the new element type automatically.

REFERENCES

[ANS98] Ansys Elements Reference, 10th ed., ANSYS Release 5.5, ANSYS, Inc, September 1998.

[BAL04] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang, PETSc Users Manual, ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[BEA97] M. W. Beall and M. S. Shephard, "A general topology-based mesh data structure," Int. J. Numer. Meth. Engr. 1997; 40(9): 1573-1596.

[HUG87] Thomas J. R. Hughes, "The Finite Element Method, Linear Static and Dynamic Finite Element Analysis," Prentice-Hall, 1987.

[ITA08] The Terascale Simulation Tools and Technology (TSTT) Center, http://www.tstt-scidac.org/.

[MET08] METIS Family of Multilevel Partitioning Algorithms, http://www-users.cs.umn.edu/~karypis/metis/index.html.

[MOA08] MOAB – A Mesh-Oriented datABase,

http://trac.mcs.anl.gov/projects/ITAPS/wiki/MOAB, 2008.

[PAT08] MSC.Patran Reference Manual, Part 4: Finite Element Modeling, MSC.Software, www.mscsoftware.com, 2008.

[SCH92] Larry A. Schoof, Victor R. Yarberry, "EXODUS II: A Finite Element Data Model," SAND92-2137, Sandia National Laboratories, Albuquerque, NM, September 1994, http://endo.sandia.gov/SEACAS/Documentation/exodusII.pdf.

[TAU04] Timothy J. Tautges, Ray E. Meyers, Karl Merkley, Clint Stimpson, Corey Ernst, "MOAB: A Mesh-Oriented Data Base," Sandia National Laboratories Report SAND2004-1592, Sandia National Laboratories, Albuquerque, NM, 2004.

Table 1: Enumerated types and miscellaneous functions

Type/Function Name	Description
MBEntityType	Enumerated type whose values are types recognized by MBCN.
operator++	Pre- and post-fix operators.
GetBasis	Get the basis of the numbering system.
SetBasis	Set the basis of the numbering system.

Table 2: Basic data functions

Function Name	Description
EntityTypeName	Human-readable name of an entity type.
Dimension	Topological dimension of entity type.
VerticesPerEntity	Return the number of (corner) vertices contained in the specified type.
NumSubEntities	Return the number of subentities of the specified dimension bounding the entity.
SubEntityType	Return the type of a particular subentity.
SubEntityConn	Return the connectivity of the specified subentity.

Table 3: Data evaluation functions

Function Name	Description
AdjacentSides	For a specified set of facets of given dimension, return the intersection or union of all facets of specified target dimension adjacent to those facets.

Table 4: Functions operating on application data

Function Name	Description
SideNumber	Return the facet index represented in the input subentity connectivity in the input parent entity connectivity array.
ConnectivityMatch	Given two connectivity arrays, determine whether or not they represent the same entity.

Table 5: Functions involving higher-order nodes

Function Name	**Description**
HasMidEdgeNodes	Return whether an entity type has mid-edge nodes for the specified total number of nodes.
HasMidFaceNodes	Return whether an entity type has mid-face nodes for the specified total number of nodes.
HasMidRegionNodes	Return whether an entity type has mid-volume nodes for the specified total number of nodes.
HasMidNodes	Return whether an entity type has mid-nodes on edges, faces, and regions for the specified total number of nodes.
HONodeIndex	For an entity with specified type and number of vertices (corner + higher-order) and a specified subfacet dimension and index, return the expected index of the higher-order node resolving that entity.
HONodeParent	For an entity with specified type and vertex array (corner + higher-order) and a specified node, return the dimension and index of the parent subentity resolved by that node.